CYBL10x7x, CY8C4128_BL, CY8C4248_BL (256K), CY8C4246_L, CY8C4247_L, CY8C4248_L

Programming Specifications

Document No. 001-96666 Rev. *C

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com

Copyrights

Contents

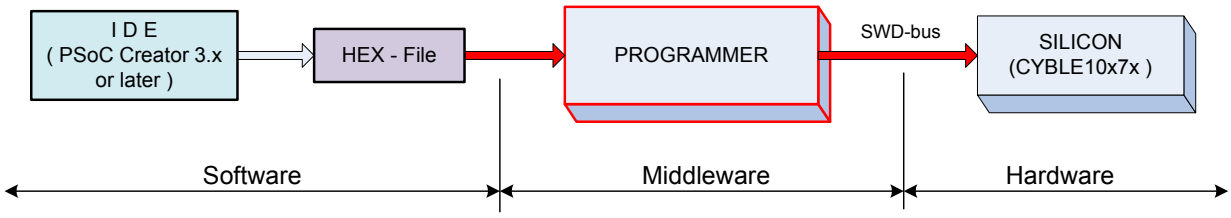
1. Introduction

This programming specifications document gives the information necessary to program the nonvolatile memory of the PSoC 4 BLE 256K and PSoC 4200-L devices. This specification describes the communication protocol required for access by an external programmer, explains the programming algorithm, and gives the electrical specifications of the physical connection. The programming algorithms described in the following sections are compatible for all devices mentioned in the title. The document will use “Target” as the generic reference to all families mentioned in the title.

1.1 Programmer

A programmer is a hardware-software system that stores a binary program (hex file) in the device's program (flash) memory. The programmer is an essential component of the engineer's prototyping environment or an integral element of the manufacturing environment (mass programming). Figure 1-1 illustrates a high-level view of the development environment.

Figure 1-1. Programmer in Development Environment



In the manufacturing environment, the integrated development environment (IDE) block is absent because its main purpose is to produce a hex file. As shown in Figure 1-1, the programmer performs three functions:

- Parses the hex file and extracts the necessary information
- Interfaces with the silicon as a serial wire debug (SWD) master
- Implements the programming algorithm by translating the hex data into SWD signals

The structure of the programmer depends on its exploiting requirements. It can be software- or firmware-centric.

In a software-centric structure, the programmer's hardware works as a bridge between the protocol (such as USB) and SWD. An external device (software) passes all SWD commands to the hardware through the protocol. The bridge is not involved in parsing the hex file and programming algorithm. This is the task of the upper layer (software). Examples of such programmers are the Cypress MiniProg3 and TrueTouchBridge.

A firmware-centric structure is an independent hardware design in which all the functions of the programmer are implemented in one device, including storage for the hex file. Its main purpose is to act as a mass programmer in manufacturing.

This document does not include the specific implementation of the programmer. It focuses on data flow, algorithms, and physical interfacing. Specifically, it covers the following topics, which correspond to the three functions of the programmer:

- Data to be programmed
- Interface with the chip
- Algorithm used to program the target device

1.2 Target Family Overview

The target family is based on the ARM Cortex-M0 processor core (48 MHz). This device family leverages the ARM debug interface for programming and debugging operations. It supports only SWD programming protocols; it does not support the JTAG interface.

The nonvolatile subsystem of the silicon consists of a flash memory system with a maximum of up to 256 KB. The flash memory system stores the user's program and silicon's protection information.

The part can be programmed after it is installed in the system by way of the SWD interface (in-system programming). The programming frequency ranges from 1.5 MHz to 14.0 MHz.

This document focuses on the specific programming operations without referencing the silicon architecture. Many important topics are detailed in the appendices. Other device-specific information can be found in the PSoC4 BLE 256K datasheet (001-94624) and the PSoC 4200-L datasheet (001-91686).

This document includes four appendices:

- Appendix A: Chip-Level Protection
- Appendix B. Intel Hex File Format
- Appendix C: Serial Wire Debug Protocol
- Appendix D: Timing Specifications of the SWD Interface

Document Revision History

Document Title: CYBL10x7x, CY8C4128_BL, CY8C4248_BL (256K), CY8C4246_L, CY8C4247_L, CY8C4248_L Programming Specifications

Document Number: 001-96666

Revision	Issue Date	Origin of Change	Description of Change
**	03/12/2015	ANDI	New specification
*A	05/19/2015	ANDI	Added support for PSoC 4200-L family devices. Added “Target” as a generic reference to all families mentioned in this document.
*B	08/30/2016	STPP	Removed “SROM_CMD_SET_IMO_48_MHz” from the Table 4.1. Removed IMO setting for 48 MHz from “Pseudocode - Step1. Acquire Chip”.
*C	05/03/2017	AESATMP8	Updated logo and Copyright.

2. Required Data

This chapter describes the information that the programmer must extract from the hex file to program the Target silicon.

2.1 Hex File Origin

Customers will use PSoC Creator to develop their projects. After development is completed, the nonvolatile configuration of the silicon is saved in the file. Only three records in this file actually target the flash memory:

- User's program (code)
- Flash row-level protection
- Chip-level protection

Other records are auxiliary and are used to keep the integrity of the programming flow.

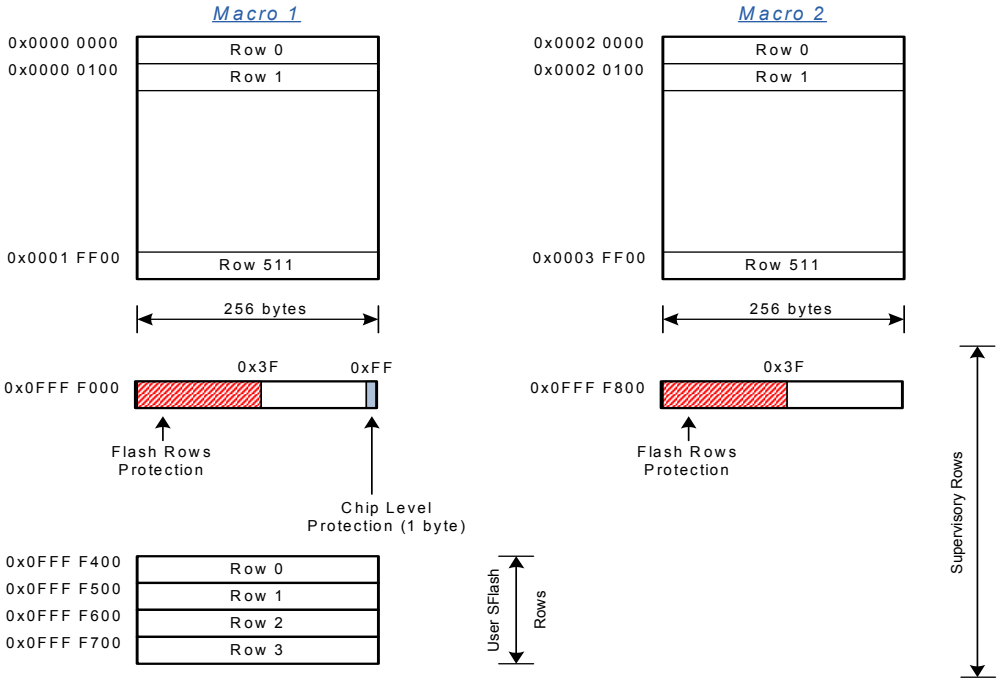
2.2 Nonvolatile Subsystem

The flash memo is organized into two macros of 128 KB each. There are 512 rows in the macro, each consisting of 256 bytes. The programming granularity is one row at a time.

In addition to the users' rows, the flash macros contain supervisory rows, which store:

- Row-level protection bits
- Chip-level protection byte (only in macro 1)
- Application-specific Information (up to four rows and only in macro 1) - User Supervisory Flash (SFlash)

Figure 2-1. Nonvolatile Subsystem



User SFlash rows can be used by the application to store arbitrary data. However, the primary intent is to store the Unique Bluetooth Address of the device. Since these rows are not part of the hex file, their programming is optional. During mass production, a vendor should define the process to guarantee for each device the uniqueness of the programmed Bluetooth address.

For the User Flash, the maximum number of rows is taken into account during programming and it depends only on the part's flash size. The formulae are as follows:

$L = 256$ – row size in bytes

$N = \frac{FlashSize}{L}$ – total number of rows

$K = \left\lfloor \frac{N}{512} \right\rfloor$ – total number of macros

The flash memory is mapped directly to the CPU's address space starting from 0x00000000. Therefore, the firmware or external programmer can read its content directly from the given address.

The flash row-level protection is a feature to write-protect the user's flash with a granularity of one row. The flash row-level protection settings prevent rows from being written but do not prevent a row's data from being read.

Each user's row in the macro is associated with one protection bit. For this reason, the maximum number of protection bits for each macro is 512. The corresponding number of bytes per macro is calculated as follows:

$ProtectionSize = \frac{512}{8} = 64$ – bytes per macro.

A bit value of 0 means that the row is unprotected and a value of 1 means the row is protected.

The last type of nonvolatile information in flash is chip-level protection. This consists of one byte that restricts access to the chip's resources (register, SRAM, and flash) by an external programmer or debugger. For example, in PROTECTED mode, the programmer cannot read or write either flash or SRAM; in KILL mode, the SWD interface is locked in silicon and the chip cannot be reprogrammed. The chip-level protection setting is programmed along with the flash row-level protection into the supervisory row of the "macro" (see Figure 2-1 on page 7). Its offset in the supervisory row is 0x7F. For more information about chip-level protection, see Appendix A: Chip-Level Protection on page 36.

2.3 Organization of the Hex File

The hexadecimal (hex) file is a medium to describe the nonvolatile configuration of the project. It is the data source for the programmer.

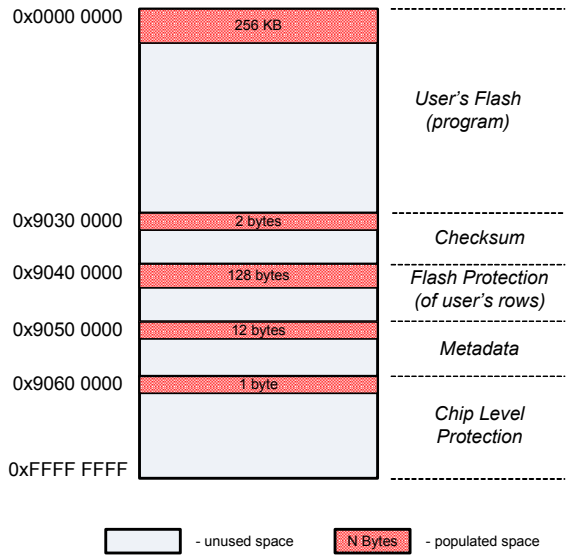
The hex file for the Target family follows the Intel Hex File format. Intel's specification is very generic and defines only some types of records that can make up the hex file. The

specification allows customizing the format for any possible silicon architecture. The silicon vendor defines the functional meaning of the records, which typically varies for different chip families. See Appendix B: Intel Hex File Format on page 38 for details of the Intel Hex File format.

The Target family defines five types of data sections in the hex file: user flash, checksum, flash protection, metadata, and chip-level protection. See Figure 2-2 to determine the allocation of these sections in the address space of the Intel hex file.

The address space of the hex file does not map to the physical addresses of the CPU (other than the user's flash, which is an unintentional coincidence). The programmer uses hex addresses (see Figure 2-2) to read sections from the hex file into its local buffer. Later, this data is programmed (translated) into the corresponding addresses of the silicon.

Figure 2-2. Hex File Organization for Target Family



0x0000 0000 – User's Flash (256 KB): This is the user's program (code) that must be programmed. The size of this section matches the flash size of the target part. The programmer can either read all of this section at once or gradually by 256-byte blocks. The programming of the flash is carried out on the row on the basis of 128 bytes for each request.

0x9030 0000 – Checksum (2 bytes): This is the checksum of the entire user flash section—the arithmetical sum of every byte in the user's flash. Only two least significant bytes (LSB) of the result are saved in this section, in big-endian format (most significant byte (MSB) first). This must be used by the programmer to check the integrity of the hex file and to verify the quality of the programming. In this context, "integrity" means that the checksum and user's flash sections must be correlated in this file. At the end of programming, the checksum of flash (two LSBs) is compared to the checksum from the hex file.

0x9040 0000 – Flash Protection (128 bytes): This data is programmed into supervisory rows of the flash macros (see Figure 2-1 on page 7). Every bit defines the write-protection setting for the corresponding user row. The number of bytes to be read from this section depends on the flash size.

$$Protection\ Size = Flash\ Size / Row\ Size / 8$$

Therefore, for a 256-KB part, flash protection consists of 128 bytes.

0x9060 0000 – Chip-level Protection (1 byte): This section represents chip-level protection of the programmed part (see Figure 2-1 on page 7). For more information, see Appendix A: Chip-Level Protection on page 36.

0x9050 0000 – Metadata (12 bytes): This section contains data that is not programmed into the target device. Instead, it is used to check data integrity of the hex file and the silicon ID of the target device. Table 2-1 lists the fields in this section.

Table 2-1. Meta Data in Hex File

Offset	Data Type	Length in Bytes
0x00	Hex file version	2 (big-endian)
0x02	Silicon ID	4 (big-endian)
0x06	Reserved	1
0x07	Reserved	1
0x08	Internal use	4

- **Hex file version**: This 2-byte field in Cypress's hex file defines its version (or type). The version for the Target family is "2". The programmer should use this field to make sure this file corresponds to the Target device, or to select the appropriate parsing algorithm if the file supports several families.
- **Silicon ID**: This 4-byte field represents the ID of the target silicon. During programming, the ID of the acquired device is compared with the content of this field. To start programming, these fields must match. Cypress does not guarantee reliable programming (or data retention) if third-party programmers ignore this condition.
- **Reserved**: Not used by the Target family.
- **Internal Use**: This 4-byte field is used internally by the PSoC Programmer software. Because it is not related to actual programming, this field should be ignored by third-party vendors.

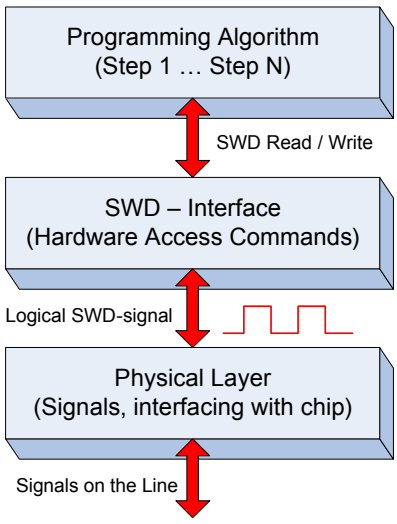
3. Communication Interface

This chapter explains the low-level details of the communication interface.

3.1 The Protocol Stack

Figure 3-1 illustrates the stack of protocols involved in the programming process. The programmer must implement both hardware and software components.

Figure 3-1. Programmer's Protocol Stack



The Programming Algorithm protocol, the topmost protocol, implements the whole programming flow in terms of atomic SWD commands. It is the most solid and fundamental part of this specification. For more information on this algorithm, see Chapter 4: Programming Algorithm on page 14.

The SWD Interface and physical layer are the lower layer protocols. Note that the physical layer is the complete hardware specification of the signals and interfacing pins, and includes drive modes, voltage levels, resistance, and other components. Upper protocols are logical and algorithmic levels.

The purpose of the SWD interface layer is to act as a bridge between pure software and hardware implementations. The Programming Algorithm protocol is implemented completely in software; its smallest building block is the SWD command. The whole programming algorithm is the meaningful flow of these blocks. The SWD interface helps to isolate the programming algorithm from hardware specifics, which makes the algorithm reusable. The SWD interface must transform the software representation of these commands into line signals (digital form).

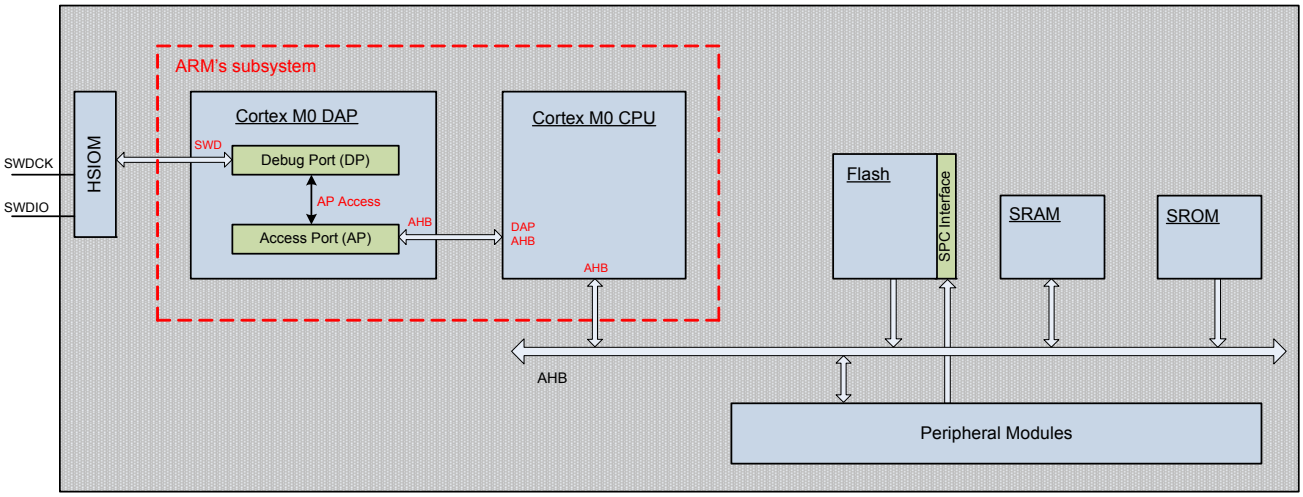
3.2 SWD Interface

The SWD interface uses the SWD protocol developed by ARM. The Target silicon integrates the standard Cortex-M0 debug access port (DAP) block provided by ARM. Therefore, it complies with the ARM specification *ARM Debug Interface v5. Architecture Specification*. The CYBKE10x6x silicon does not support the JTAG interface.

Figure 3-2 on page 10 shows the top-level architecture of the silicon. It includes the debug interface, CPU subsystem, memory, and periphery. The standard ARM modules are outlined in red. The following acronyms are used in this figure:

- HSIOM – High-speed I/O matrix
- DAP – Debug access port
- AHB – Advanced high-performance bus
- SPC – System performance controller

Figure 3-2. Top-Level Silicon Architecture



The SWD interface (ARM) defines only two digital pins to communicate with an external programmer or a debugger. The SWDCK and SWDIO pins are sufficient for bidirectional, semi-duplex data exchange.

Only three types of SWD commands can appear on the bus: Read, Write, and Line Reset. The Line Reset command is used only once during programming to establish a connection with the device. The Read and Write commands compose the rest of the programming flow.

The programmer can access most resources of the silicon through the SWD interface. All programming algorithms are stored in SROM; the external programmer uses its system APIs to program the flash. During programming of the flash row, the system code is executed from the SROM. It communicates with the SPC module, which “knows” how to program flash. In contrast to a write operation, reading from flash is an immediate operation that is carried out directly from the necessary address (see Figure 2-1 on page 7 for address space). Reading works on a word basis (4-byte); writing works on a row basis (256-byte).

The typical operation of the programmer is to load all necessary parameters into the SRAM (I/O registers) and request a system call from the SROM. Only the SWD Read and Write commands perform this task.

3.3 Hardware Access Commands

The Cortex-M0 DAP module, shown in Figure 3-2, supports three types of transactions: Read, Write, and Line Reset. All are defined in the ARM specification. The APIs must be implemented by the SWD Interface layer shown in Figure 3-1 on page 9. In addition, the upper protocol, Programming Algorithm, requires two extra commands to manipulate the hardware: Power(state) and ToggleReset(). Table 3-1 lists the hardware access commands used by the software layer.

Table 3-1. Hardware Access Commands

Command	Parameters	Description
SWD_LineReset		Standard ARM command to reset the debug port (DAP). It consists of at least 50 clock cycles with data = 1, that is, with the SWDIO asserted HIGH by the programmer. Transaction must be completed at least by 1 clock with SWDIO asserted LOW. This sequence synchronizes the programmer and chip; it is a first transaction in programming flow.
SWD_Write	IN APnDP, IN addr, IN data32, OUT ack	Sends a 32-bit data to the specified register of the DAP. The register is defined by the “APnDP” (1 bit) and “addr”(2 bits) parameters. The DAP returns a 3-bit status in “ack”.
SWD_Read	IN APnDP, IN addr, OUT data32, OUT ack, OUT parity	Reads a 32-bit data from the specified register of the DAP. The register is defined by the “APnDP” (1 bit) and “addr” (2 bits) parameters. DAP returns a 32-bit data, status, and parity (control) bit of the read 32-bit word.
ToggleReset		Generates the reset signal for target device. The programmer must have a dedicated pin connected to the XRES pin of the target device.
Power	IN state	If the programmer powers the target device, it must have this function to supply power to the device.

For information on the structure of the SWD read and write packets and their waveform on the bus, see Appendix C: Serial Wire Debug (SWD) Protocol on page 39.

The SWD_Read and SWD_Write commands allow accessing registers of the Cortex-M0 DAP module from Figure 3-2 on page 10. The DAP functionally is split into two control units:

- Debug Port (DP) – Is responsible for the physical connection to the programmer or debugger.
- Access Port (AP) – Provides the interface between the DAP module and one or more debug components (such as the Cortex-M0 CPU).

The external programmer can access the registers of these access ports using the following bits in the SWD packet:

- APnDP – Select access port (0 – DP, 1 - AP).
- ADDR – 2-bit field addressing a register in the selected access port

The SWD_Read and SWD_Write commands are used to access these registers. They are the smallest transactions that can appear on the SWD bus. Table 3-2 shows the DAP registers that are used during programming.

Table 3-2. DAP Registers (in ARM notation)

Register	APnDP (1 bit)	Address (2-bit)	Access (R/W)	Full Name
IDCODE	0	2'b00	R	Identification Code Register
CTRL/STAT	0	2'b01	R/W	Control/Status Register
SELECT	0	2'b10	W	AP Select Register
CSW	1	2'b00	R/W	Control Status/Word Register (CSW)
TAR	1	2'b01	R/W	Transfer Address Register
DRW	1	2'b11	R/W	Data Read/Write Register

For more information about these registers, see the *ARM Debug Interface v5. Architecture Specification*.

3.4 Pseudocode

This document uses easy-to-read pseudocode to show the programming algorithm. The following two commands are used for the programming script:

```
Write_DAP ( Register,  Data32)
Read_DAP ( Register, OUT Data32)
```

Where the `Register` parameter is an AP/DP register defined by APnDP and address bits (see Table 3-2). The pseudocommands correspond to read or write SWD transactions. Following are some examples:

```
Write_DAP( TAR, 0x20000000)
Write_DAP( DRW, 0x12345678)
Read_DAP ( IDCODE, OUT swd_id)
```

The `Register` parameter technically can be represented as the structure in C:

```
struct DAP_Register
{
BYTE APnDP; // 1-bit field
BYTE Addr;  // 2-bit field
};
```

Then, DAP registers will be defined as:

```
DAP_Register TAR    = { 1, 1 },
             DRW    = { 1, 3 },
             IDCODE= { 0, 0 };
```

The defined Write and Read pseudocommands must be successful if both return the ACK status of the SWD transaction. For the Read transaction, the parity bit must be taken into account (corresponds to read data32 value). If the status of the trans-

action, the parity bit, or both is incorrect, the transaction must be considered to have failed. In this case, depending on the programming context, programming must terminate or the transaction must be tried again.

The implementation of Write and Read pseudocommands based on the hardware access commands SWD_Read and SWD_Write (Table 3-1 on page 10) is as follows.

```
SWD_Status Write_DAP ( Register,  data32 ) {
  SWD_Write ( Register.APnDP, Register.Addr, data32, OUT ack);
  Return ack;
}

SWD_Status Read_DAP ( Register, OUT data32){
  SWD_Read ( Register.APnDP, Register.Addr, OUT data32, OUT ack, OUT parity);
  If (ack == 3'b001){ //ACK, then check also the parity bit
    Parity_data32 = 0x00;
    For (i=0; i<32; i++) Parity_data32 ^= ((data32 >> i) & 0x01);
    If (Parity_data32 != parity) ack = 3'b111; //NACK
  }
  Return ack;
}
```

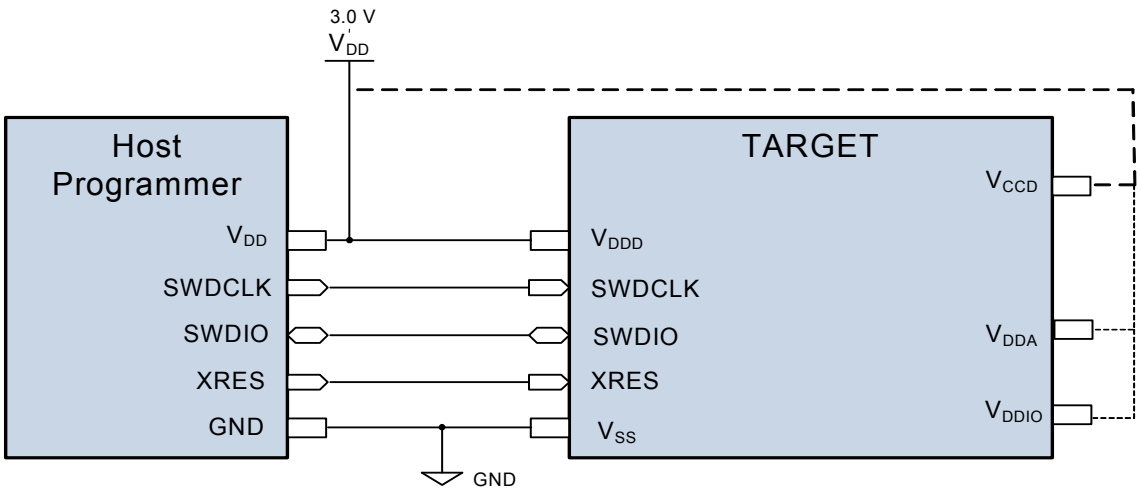
The programming code in Chapter 4: Programming Algorithm on page 14 is based mostly on the Write and Read pseudo-commands and some commands in Table 3-1 on page 10.

3.5 Physical Layer

This section describes the hardware connections between the programmer and the target device for programming. It shows the connection schematic and gives information on electrical specifications. Check the device datasheet for the actual location of SWD/Power pins on the part's package.

Check the device datasheet for the actual location of SWD/Power pins on the part's package. The generic connection between the target and programmer is shown in Figure 3-3.

Figure 3-3. Connection Schematic of Programmer



Only five pins are required to communicate with the chip. Note that the SWDCLK and SWDIO pins are only required by the SWD protocol. The silicon requires an additional XRES pin that is not related to the ARM standard. It is used to reset the part as a first step in a programming flow.

You can program a chip in either Reset or Power Cycle mode. The mode defines only the first step—how to reset the part—in the programming flow. The rest of the steps are identical (SWD traffic).

- **Reset mode**: To start programming, the host toggles the XRES line and then sends SWD commands (see Table 3-1 on page 10). The power on the target board can be supplied by the host or by an external power adapter (the V_{DD} line can be optional).
- **Power Cycle mode**: To start programming, the host powers on the target and then starts sending the SWD commands. The XRES line is not used.

It is recommended that the programmer uses all five pins and supports at least Reset mode programming. The Power Cycle mode support is optional.

Table 3-3. Programming Mode

Mode	Necessary Pins	Unused Pins	Use Cases
Reset	V_{DD} (optional) GND XRES SWDCLK SWDIO	V_{DD} (if self-powered)	The board can be self-powered (V_{DD} is not needed). The board consumes too much current, which the programmer cannot supply (V_{DD} is not needed). The 5-pin case: The host supplies power and toggles XRES (this is the most popular programming method).
Power Cycle	V_{DD} GND SWDCLK SWDIO	XRES	The only use case: The XRES pin is not available on the part's package, so Power Cycle mode is the only way to reset a part. This is not applicable to the CYBL10x7x family, in which every package has an XRES pin. For this reason, Reset is the recommended mode. Some third-party SWD masters can use this mode if they do not implement the XRES line but can supply power (power on/off).

Table 3-4. Target Pin Names and Requirements

Target Pin Name	Function	External Programmer Drive Modes
V_{DDD}	Digital power supply Input (1.71 V–5.5 V)	Positive voltage – powered by external power supply or by programmer.
V_{SS}	Power supply return	Low resistance ground connection. Connect to circuit ground.
XRES	External active low reset input.	Output: Drive CMOS levels
SWDCLK	SWD clock input (1.5 MHz–14 MHz)	Output: Drive CMOS levels
SWDIO	SWD data line - bidirectional	Output: Drive CMOS levels Input: Read CMOS levels in HI-Z mode
V_{DDA}	Analog power supply input (1.71 V–5.5 V)	Depending on the board configuration, this power can be supplied from a V_{DDD} source, which must be in the range 1.9 V–5.5 V. If it is lower, then a separate voltage source is required for analog circuits. Depending on the board's configuration, this power can be supplied from a V_{DDD} source or own analog power supply. Check the silicon's datasheet for exact power pins configuration. For example, for the BLE family V_{DDA} can be powered from V_{DDD} if it is in range of 1.9 V–5.5 V (if V_{DDD} is lower, then own V_{DDA} source is required).
VDDIO	I/O pins power supply	Available only for PSoC 4200-L series. Typically connected to V_{DDD} (application-specific).

The SWD timing specifications are described in the datasheet (001-94624 or 001-91686).

.

4. Programming Algorithm

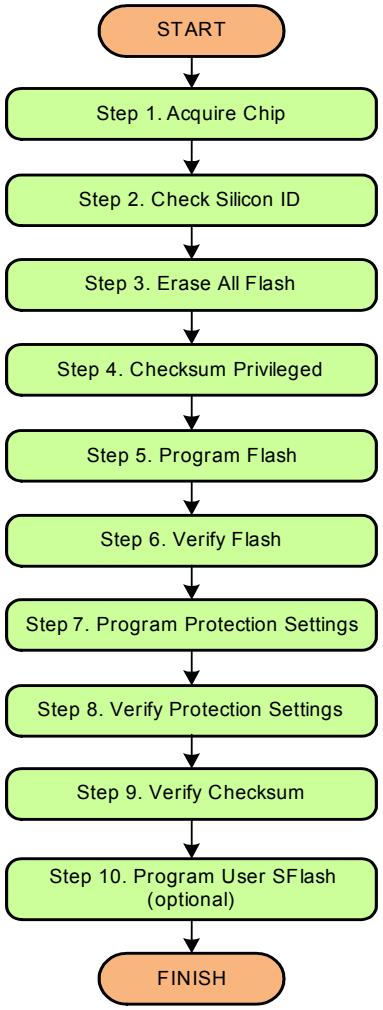
This chapter describes in detail the programming flow of the Target device. It starts with a high-level description of the algorithm and then describes every step using pseudocode. All code is based on upper-level subroutines composed of atomic SWD instructions (see "Pseudocode" on page 11). These subroutines are defined in "Subroutines Used in the Programming Flow" on page 15. The ToggleReset() and Power() commands are also used (see Table 3-1 on page 10).

4.1 High-Level Programming Flow

Figure 4-1 shows the sequence of steps that must be executed to program the Target device. These steps are described in detail in the following sections. All the steps in this programming flow must be completed successfully for a successful programming operation. The programmer should stop the programming flow if any step fails. In addition, in pseudocode, it is assumed that the programmer checks the status of each SWD transaction (Write_DAP, Read_DAP, WriteIO, ReadIO). This extra code is not shown in the programming script. If any of these transactions fails, then programming must be aborted.

Flash programming in the Target family is implemented using the SROM APIs. The external programmer puts the parameters into the SRAM (or registers) and requests system calls, which in turn perform flash updates.

Figure 4-1. High-Level Programming Flow of Target Device



4.2 Subroutines Used in the Programming Flow

The programming flow includes some operations that are used in all steps. Eventually, the programming code will look compact and easy to read and understand. Besides that, most of the registers and frequently used constants are named and referred from the pseudocode.

Table 4-1. Constants Used in the Programming Script

Constant Name	Value	Description
Address Space of CPU		
CPUSS_SYSREQ	0x40100004	System request register used to make system requests to SROM code; system requests transition from User mode to Privileged mode
CPUSS_SYSARG	0x40100008	System request argument register used to make system requests to SROM code
TEST_MODE	0x40030014	Test mode control register used to enter the chip into Programming mode (Test mode)
SRAM_PARAMS_BASE	0x20000100	SRAM address where the parameters for SROM requests will be stored.
SFLASH_MACRO_0	0x0FFFF000	Location of the flash protection settings in the flash macro 0.
SFLASH_MACRO_1	0x0FFFF800	Location of the flash protection settings in the flash macro 1.
SFLASH_CPUSS_PROTECTION	0x0FFFF0FC	Location of chip-level protection in the flash macro (actual byte located at 0x0FFFF0CF, but must read whole 32-bit word).
SROM Constants		
SROM_KEY1	0xB6	Parameter of SROM call
SROM_KEY2	0xD3	Parameter of SROM call
SROM_SYSREQ_BIT	0x80000000	Mask of SYSREQ bit in CPUSS_SYSREQ register, which starts the execution of the SROM command
SROM_PRIVILEGED_BIT	0x10000000	Mask of PRIVILEGED bit in CPUSS_SYSREQ register, which indicates whether the system is in Privileged mode (SROM command running) or User mode.
SROM_STATUS_SUCCEEDED	0xA0000000	Successful status of the system request (SROM command).
SROM_STATUS_FAILED	0xF0000000	Fail status of the system request (SROM command).
SROM Requests		
SROM_CMD_GET_SILICON_ID	0x00	Reads the silicon ID of the target device.
SROM_CMD_LOAD_LATCH	0x04	Loads data into the volatile buffer (before writing into flash).
SROM_CMD_PROGRAM_ROW	0x06	Programs data into the flash row (from the volatile buffer).
SROM_CMD_ERASE_ALL	0x0A	Erases all the user's flash and flash protection settings from the supervisory rows
SROM_CMD_CHECKSUM	0x0B	Verifies the checksums of all flash contents (user and privileged rows)
SROM_CMD_WRITE_PROTECTION	0x0D	Writes flash protection and chip-level protection
SROM_CMD_WRITE_SFLASH_ROW	0x18	Writes User SFlash Row. Valid row range is [0..3].
Chip -Level Protection		
CHIP_PROT_VIRGIN	0x00	VIRGIN mode, used by Cypress only.
CHIP_PROT_OPEN	0x01	OPEN mode in which the chip is shipped to customers
CHIP_PROT_PROTECTED	0x02	PROTECTED mode, which can be set by the customer
CHIP_PROT_KILL	0x04	KILL mode, which can be set by the customer (irreversible)

Table 4-2. Subroutines Used in Programming Flow

Subroutine	Description
bool WriteIO(addr32, data32)	Writes a 32-bit data into the specified address of the CPU address space. Returns "true" if all SWD transactions succeeded (ACKed).
bool ReadIO(addr32, OUT data 32)	Reads a 32-bit data from the specified address of the CPU address space. Note that the actual size of the read data (8, 16, 32 bits) depends on the setting in the CSW register of DAP (see Table 3-2). By default, all accesses are 32 bits long. Returns "true" if all SWD transactions succeeded (ACKed).
bool PollSROMStatus()	Waits until the SROM command is completed and then checks its status. Timeout is 1 second. Returns "true" (success) if the command is completed and its status is successful; otherwise, returns "false".

The implementation of these subroutines follows. It is based on the pseudocode and registers defined in “Hardware Access Commands” on page 10 and “Pseudocode” on page 11. It uses the constants defined in this chapter.

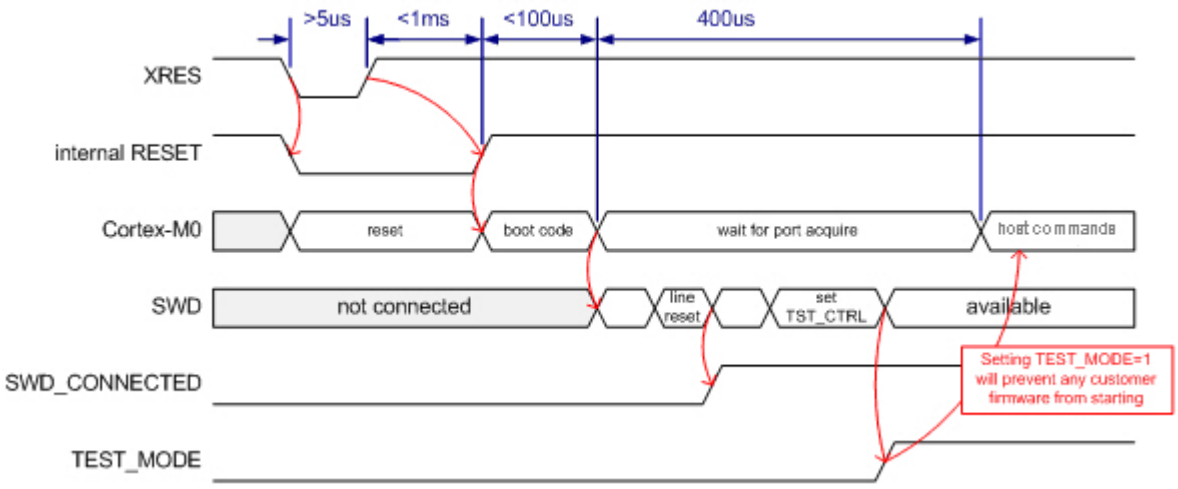
The pseudocode is similar to C-style notation.

```
 // WriteIO Subroutine
bool “WriteIO” ( addr32, data32 )
{
ack1 = Write_DAP (TAR, addr32);
ack2 = Write_DAP (DRW, data32);
return (ack1 == 3’b001) && (ack2 == 3b’001);
}

// “ReadIO” Subroutine
bool ReadIO ( addr32, OUT data32 )
{
ack1 = Write_DAP (TAR, addr32);
ack2 = Read_DAP (DRW, OUT data32);
ack3 = Read_DAP (DRW, OUT data32);
return (ack1 == 3’b001) && (ack2 == 3b’001) && (ack3 == 3b’001);
}

// “PollSROMStatus” Subroutine
bool PollSROMStatus()
{
do{
ReadIO (CPUSS_SYSREQ, OUT status);
Status &= (SROM_SYSREQ_BIT | SROM_PRIVILEGED_BIT);
}while ((status != 0) && (time_elapsed < 1 sec));

if (time_elapsed >= 1 sec ) return false; // timeout

ReadIO (CPUSS_SYSARG, OUT statusCode);
if ((statusCode & 0xF0000000) != (SROM_STATUS_SUCCEEDED));
return false; // SROM command failed
else
return true; // SROM command succeeded
}
```

4.3 Step 1 – Acquire Chip

The first step in programming the Target device is to enter it into Test mode (or Programming mode). This is a special mode in which the CPU is controlled by the external programmer, which also can access other system resources such as SRAM and registers. This step has strict timing requirements that the host must meet to enter Test mode successfully. Figure 4-2 shows the timing diagram of entering Test mode.

Figure 4-2. Timing Diagram of Entering Test Mode

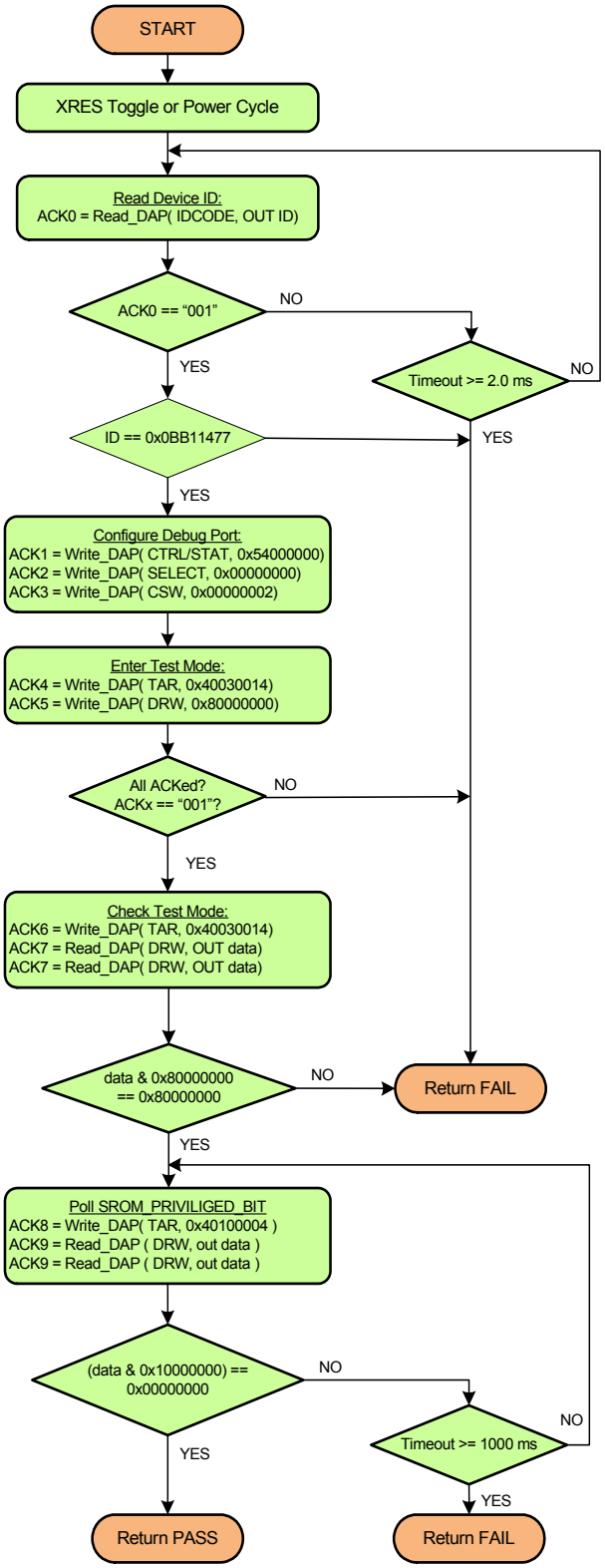


This diagram details the chip's internal signals while entering Test mode. Everything starts from toggling the XRES line (or applying power), so the chip enters Internal Reset mode. After that, the system boot code starts execution from the SROM. When completed, the CPU waits during a 400-µs time frame for a special connection sequence on the SWD port. If, during this time, the host sends the correct sequence of SWD commands, the CPU enters Test mode. Otherwise, it starts the execution of the user's code.

The times of internal reset (<1 ms) and boot code (<100 µs) are not specified exactly. Because they depend on the CPU clock and the size of the code, they can vary in different revisions of the chip.

In this case, the recommended way to enter Test mode is to start sending an acquire sequence right after XRES is toggled (or power is supplied in Power Cycle mode). This sequence is sent iteratively until it succeeds; that is, all SWD transactions are ACKed and all conditions are met. Figure 4-3 on page 18 shows the implementation of the Acquire Chip procedure. It is detailed in terms of the SWD transaction. Note that the recommended minimum frequency of the programmer is 1.5 MHz, which meets the timing requirement of this step (400 µs).

Figure 4-3. Flow Chart of the Acquire Chip Step



Pseudocode – Step 1. Acquire Chip

```
//-------------------------------------------------------------------------
// Reset Target depending on acquire mode – Reset or Power Cycle
If (AcquireMode == "Reset") ToggleXRES(); // Toggle XRES pin, target must be powered.
Else If (AcquireMode == "Power Cycle") PowerOn();// Supply power to target.

//Execute ARM's connection sequence – acquire SWD-port.
Do
{
SWD_LineReset();
ack = Read_DAP ( IDCODE, out ID);

}While ((ack != 3b'001) && time_elapsed < 1.5 ms); //for PowerCycle timeout must be
   //longer. For example ~30 ms.
If (time_elapsed >= 1.5 ms) Return FAIL;

If (ID != 0x0BB11477) Return FAIL; //SWD ID of Cortex-M0 CPU.

//Initialize Debug Port
Write_DAP (CTRL/STAT, 0x54000000);
Write_DAP (SELECT, 0x00000000);
Write_DAP (CSW, 0x00000002);

//Enter CPU into Test Mode
WriteIO (TEST_MODE, 0x80000000); //Set test_mode bit in TEST_MODE reg from CPU space
ReadIO  (TEST_MODE, out status);

if ((status & 0x80000000) != 0x80000000) Return FAIL;

//Poll SROM_PRIVILEGED_BIT in CPUSS_SYSREQ register
Do
{
ReadIO (CPUSS_SYSREQ, out status);
status &= SROM_PRIVILEGED_BIT;
} While ((status != 0x00000000) && time_elapsed < 1000 ms)

If (time_elapsed >= 1000 ms) Return FAIL;

Return PASS;
```

4.4 Step 2 – Check Silicon ID

This step is required to verify that the acquired device corresponds to the hex file. It reads the ID from the hex file and compares it with the ID obtained from the target.

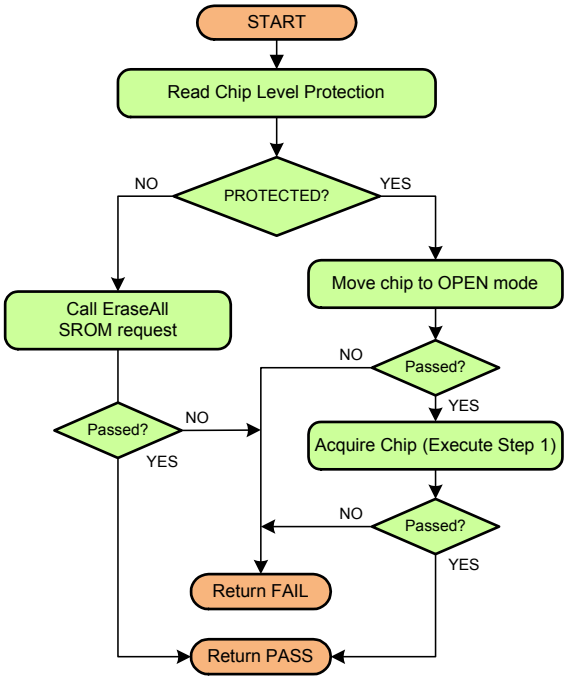
Pseudocode – Step 2. Check Silicon ID

```
//-------------------------------------------------------------------------------
// Read "Silicon ID" from hex file, 4 bytes from address 0x9050 0002 (big endian).
// HEX_ReadSiliconID() must be implemented.
HexID = HEX_ReadSiliconID();

// Read "Silicon ID" from the target using SROM request
Params = (SROM_KEY1 << 0) +    //KEY1
        ((SROM_KEY2+SROM_CMD_GET_SILICON_ID) << 8);    //KEY2

WriteIO (CPUSS_SYSARG, Params); // Write parameters
WriteIO (CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_GET_SILICON_ID);//Request SROM call

status = PollSromStatus();
If (!status) Return FAIL;

//Read 32-bit ID from the registers
ReadIO( CPUSS_SYSARG, out part0);
ReadIO( CPUSS_SYSREQ, out part1);

siliconID[0] = (part0 >> 8) & 0xFF;
siliconID[1] = (part0 >> 0) & 0xFF;
siliconID[2] = (part0 >> 16)& 0xFF;
siliconID[3] = (part1 >> 0) & 0xFF;

//Compare IDs from the hex and from the target
For ( i = 0; i < 4; i++)
{
If ( i == 2 ) //Ignore Revision ID
{
    Continue;
}
If ( siliconID[i] != hexID[i] ) Return FAIL;
}

Return PASS;
```

4.5 Step 3 – Erase All Flash

Before programming the flash, it must be erased. This step erases all user rows and the corresponding flash protection. It also moves chip-level protection to the OPEN state (if it was in PROTECTED mode, see Appendix A:Chip-Level Protection on page 36). Figure 4-4 shows the algorithm of the Erase All step.

Figure 4-4. Flow Chart of the Erase All Step



Pseudocode – Step 3. Erase All Flash

```
//-------------------------------------------------------------------------------
// Read Chip Level Protection using SROM call
Params = (SROM_KEY1 << 0) + // KEY1
         ((SROM_KEY2 + SROM_CMD_GET_SILICON_ID) << 8); // KEY2

WriteIO( CPUSS_SYSARG, Params);  //Write params in CPUSS_SYSARG
WriteIO( CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_GET_SILICON_ID);//Request SROM call

Status = PollSromStatus();
If (!Status)) Return FAIL;

ReadIO( CPUSS_SYSREQ, out data); // read result
chipProt = (byte)(data  >> 12);

// Check current protection mode
If (chipProt == CHIP_PROT_PROTECTED) // PROTECTED
{
// Move chip to OPEN mode
Params = (SROM_KEY1 << 0) +//KEY1
               ((SROM_KEY2 + SROM_CMD_WRITE_PROTECTION) << 8) +//KEY2
                (0x01 << 16) + //OPEN mode
                (0x00 << 24);//Flash Macro 0

       WriteIO( CPUSS_SYSARG, Params);//Write params in CPUSS_SYSARG
       WriteIO( CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_WRITE_PROTECTION);
```

```
        status = PollSromStatus();
        if (!status) return FAIL;

// Re-acquire chip here to boot it in OPEN mode
// Execute Now: "Step 1 - Acquire Chip"

// Check result of re-acquire
If (!status) Return FAIL;
}
Else  // OPEN (CHIP_PROT_OPEN)
{
Params = (SROM_KEY1 << 0) +//KEY1
              ((SROM_KEY2+SROM_CMD_ERASE_ALL) << 8);//KEY2

        WriteIO( SRAM_PARAMS_BASE + 0x00, Params); //Write params in SRAM
        WriteIO( CPUSS_SYSARG, SRAM_PARAMS_BASE);  //Set location of parameters
        WriteIO( CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_ERASE_ALL); //Request SROM call

        status = PollSromStatus();
If (!status) Return FAIL;
}
Return PASS;
//-------------------------------------------------------------------------------
```

4.6 Step 4 – Checksum Privileged

After the user's flash is erased, its checksum must be 0x00. However, the Checksum(All) API, which is used in "Step 9 – Verify Checksum" on page 31, also calculates the checksum of the privileged rows (not only the user rows). Therefore, it is necessary to find the checksum of the privileged rows when all the user rows are erased. The result of this operation is needed only to calculate the proper checksum of the user's flash in "Step 9 – Verify Checksum" on page 31. That checksum is calculated according to the following formula:

Checksum_User = Checksum_Step_9 – Checksum_Step_4

The result of this step must be used in "Step 9 – Verify Checksum" on page 31. A possible alternative solution to avoid this step is to calculate the checksum of each row individually and add them. However, this method takes much longer.

Pseudocode – Step 4. Checksum Privileged

```
//-------------------------------------------------------------------------------
Params = (SROM_KEY1 << 0) +//KEY1
         ((SROM_KEY2+SROM_CMD_CHECKSUM) << 8)+//KEY2
         ((0x0000 & 0x00FF) << 16) +//Row ID[7:0]
         ((0x8000 & 0xFF00) << 16);//Row ID[15:8] - Checksum All(0x8000)

WriteIO( CPUSS_SYSARG, Params); //Write params in CPUSS_SYSARG
WriteIO( CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_CHECKSUM); //Request SROM call

status = PollSromStatus();
if (!status) return FAIL;

//Read Checksum from CPUSS_SYSARG register
ReadIO(CPUSS_SYSARG, out checksum_all);

Checksum_Privileged = (checksum_all & 0x0FFFFFFF); //28-bit checksum
return PASS;
//-------------------------------------------------------------------------------
```

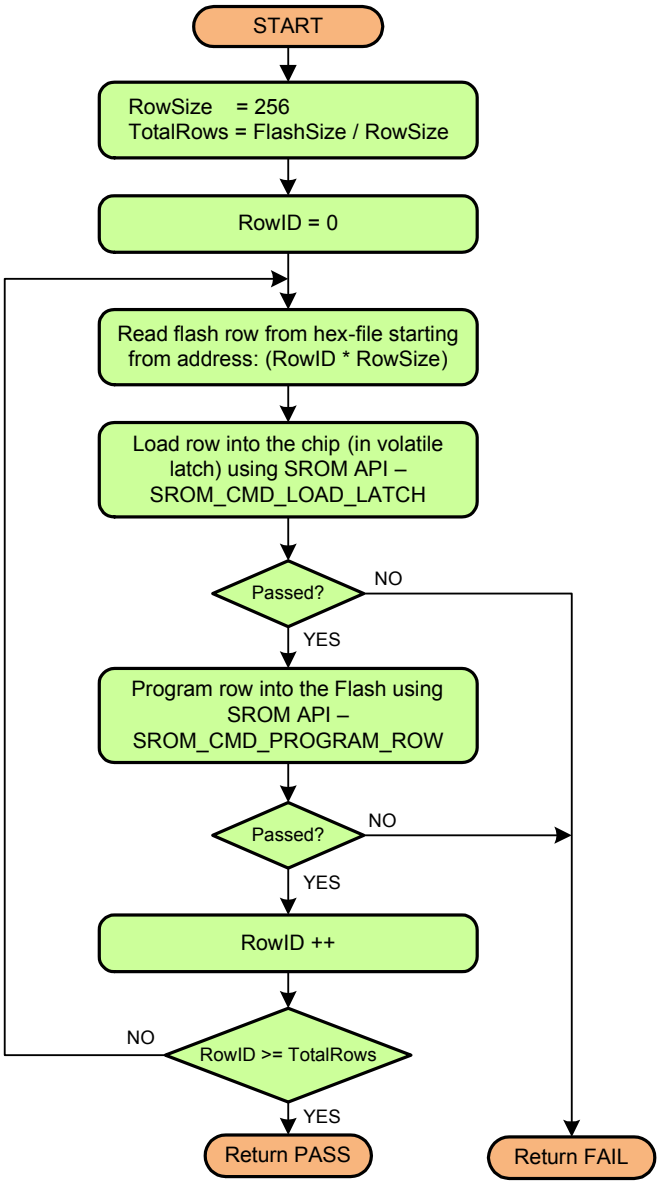
4.7 Step 5 – Program Flash

Flash memory is programmed in rows. Each row is 256 bytes long. The programmer must serially program each row individually. The source data is extracted from the hex file starting from address 0x00000000 (see Figure 2-2 on page 8). The flash size and the row size are input parameters of this step. Note that the flash size of the acquired silicon must be equal to the size of the user's code in the hex file, as verified in Step 2 by comparing the silicon IDs of the hex and the target. During programming, two SROM APIs are used:

- SROM_CMD_LOAD_LATCH – Loads the flash row into the silicon's volatile buffer.
- SROM_CMD_PROGRAM_ROW – Programs the row into flash (from the volatile buffer).

Figure 4-5 illustrates this programming algorithm.

Figure 4-5. Flow Chart of the "Program Flash" Step



Pseudocode – Step 5. Program Flash

```
//------------------------------------------------------------------------------
// Flash Size must be provided.
RowSize = 256;
TotalRows = FlashSize / RowSize;
RowsPerMacro = 512;
//Program all flash rows
for (int RowID = 0; RowID < TotalRows; RowID++)
{
    //1. Read Row data from hex
    RowHexAddress = RowSize * RowID;
    //Extract 256-byte row from the hex-file
    //from address: "RowHexAddress" into buffer - "Data".
    //HEX_ReadData() must be implemented by Programmer.
    Data = HEX_ReadData( RowHexAddress, RowSize );

    //2. Load Row to volatile buffer (latch)
    MacroID = floor ( RowID / RowsPerMacro ); //Round to largest previous integer

    Params1 = (SROM_KEY1 << 0) +//KEY1
        (SROM_KEY2 + SROM_CMD_LOAD_LATCH) << 8) +//KEY2
        (0x00 << 16);//Byte number in latch from what to write
        (MacroID << 24);       //Flash Macro ID (0 or 1)

    Params2 = (RowSize - 1); //Number of Bytes to load minus 1
    WriteIO(SRAM_PARAMS_BASE + 0x00, Params1);  //Write params in SRAM
    WriteIO(SRAM_PARAMS_BASE + 0x04, Params2); //Write params in SRAM

    // Put row data into SRAM buffer
    for (i = 0; i < RowSize; i += 4)
    {
        Params1 = (Data[i] << 0) + (Data[i + 1] << 8) +
            Data[i + 2] << 16) + (Data[i + 3] << 24);
            WriteIO(SRAM_PARAMS_BASE + 0x08 + i, Params1);//Write params is SRAM
    }

    // Call "Load Latch" SROM API
    WriteIO(CPUSS_SYSARG, SRAM_PARAMS_BASE); //Set location of parameters
    WriteIO(CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_LOAD_LATCH);//Request SROM operation

    Status = PollSromStatus();
    if (!Status) return FAIL;

    //3. Program Row - call SROM API
    Params =(SROM_KEY1 << 0) +     //KEY1
        ((SROM_KEY2+SROM_CMD_PROGRAM_ROW) << 8) +   //KEY2
        ((RowID & 0x00FF) <<  16) +    //ROW_ID_LOW[7:0]
        ((RowID & 0xFF00) << 16);    //ROW_ID_HIGH[9:8]

    WriteIO(SRAM_PARAMS_BASE+0x00, Params);     //Write params is SRAM
    WriteIO(CPUSS_SYSARG, SRAM_PARAMS_BASE);    //Set location of parameters
    WriteIO(CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_PROGRAM_ROW);//Request SROM operation

    Status = PollSromStatus();
    if (!Status) return FAIL;
}
return PASS;
```

4.8 Step 6 – Verify Flash

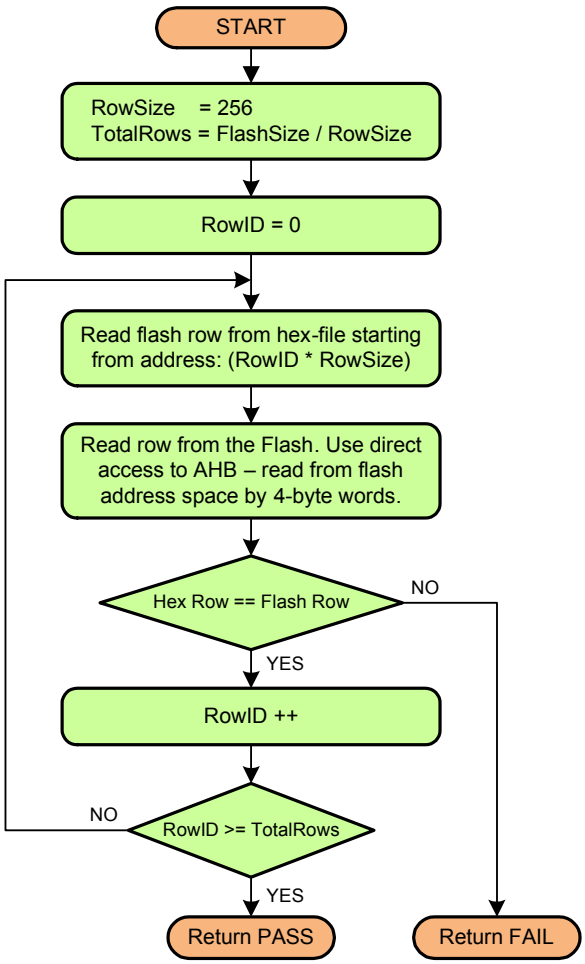
Because the checksum is verified eventually, this step is optional. It is recommended that it be kept in the programming flow for higher reliability. The checksum cannot completely guarantee that the content is written without errors.

During verification, the programmer reads a row from flash and the corresponding data from the hex file and compares them. If any difference is found, the programmer must stop and return a failure. Each row must be considered.

Reading from the flash is achieved by direct access to the memory space of the CPU. No SROM API is required; simply read from the address range 0x00000000 – 0x0003FFFF.

Figure 4-6 illustrates the verification algorithm.

Figure 4-6. Flow Chart of the “Verify Flash” Step



Pseudocode – Step 6. Verify Flash.

```
//-------------------------------------------------------------------------------
// Flash Size must be provided.
RowSize = 256;
TotalRows = FlashSize / RowSize;

//Read and Verify Flash rows
for (int RowID = 0; RowID < TotalRows; RowID++)
{
//1. Read row from hex file
       RowAddress = rowSize * rowID; //liner address of row in flash
       //Extract 256-byte row from the hex file
       //from address: "RowHexAddress" into buffer - "Data".
//HEX_ReadData() must be implemented by Programmer.
hexData = HEX_ReadData( RowAddress, RowSize );

       //2. Read row from chip
       for (i = 0; i < RowSize; i += 4)
       {
//Read flash via AHB-interface
       ReadIO( RowAddress + i, out data32);
             chipData[i + 0] = (data32 >> 0) & 0xFF;
             chipData[i + 1] = (data32 >> 8) & 0xFF;
             chipData[i + 2] = (data32 >> 16) & 0xFF;
             chipData[i + 3] = (data32 >> 24) & 0xFF;
       }

       //3. Compare them
       for (i = 0; i < RowSize; i++)
       {
       if (chipData[i] != hexData[i]) return FAIL;
       }
}
return PASS;
```

4.9 Step 7 – Program Protection Settings

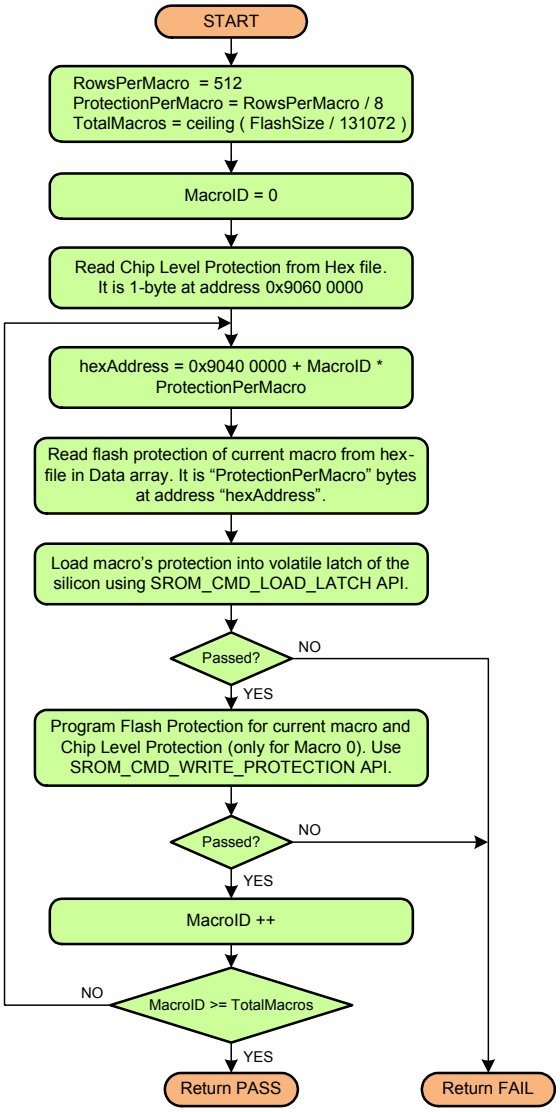
At this point, the programmer writes into the supervisory flash all protection data: row-level protection and chip-level protection. For more information, see Figure 2-1 on page 7.

The Target device can have two flash macros, each with its own supervisory rows to store the protection settings of the user's rows. Each user row occupies one bit in the protection space: 0 means unprotected; 1 means protected. This provides write/erase protection of the row. In the PROTECTED state, a row cannot be erased or written either by the firmware or by an external programmer. The protection setting can be reset only by the EraseAll() operation in Step 3, driven by the external programmer.

Chip-level protection is only 1 byte and is stored in the supervisory row of macro 0 where the flash protection data resides.

Figure 4-7 shows the algorithm of writing protection settings.

Figure 4-7. Flow Chart of the “Program Protection Settings” Step



Pseudocode – Step 7. Program Protection Settings

```
//------------------------------------------------------------------------------
// Flash Size must be provided.
RowsPerMacro = 512;
ProtectionPerMacro = RowsPerMacro / 8;

TotalMacros = ceiling ( FlashSize / 131072 ); // round to smallest following integer
//1. Read Chip Level Protection from hex-file. It is 1 byte at address 0x90600000.
//HEX_ReadChipLevelProtection() must be implemented.
ChipLevelProtection = HEX_ReadChipLevelProtection();

for (MacroID = 0; MacroID < TotalMacros; MacroID++)
{
//2. Read Protection settings of current macro from hex-file.
//It is located at address 0x9040 0000.
//HEX_ReadRowProtection() must be implemented by Programmer.
```

```
HexAddr = ProtectionPerMacro * MacroID;
Data = HEX_ReadRowProtection(HexAddr, ProtectionPerMacro);

//3. Load protection setting of current macro into volatile latch.
//This is same implementation as for "Program Flash" step.
//So this code can be moved into a separate routine - "LoadLatch(MacroID, Data)".
Params1 = (SROM_KEY1 << 0) +//KEY1
          ((SROM_KEY2 + SROM_CMD_LOAD_LATCH) << 8) +//KEY2
          (0x00 << 16)+ //Byte number in latch from what to write
          (MacroID << 24);//Flash Macro ID (0 or 1)
Params2 = (ProtectionPerMacro - 1); //Number of Bytes to load minus 1

WriteIO(SRAM_PARAMS_BASE + 0x00, Params1);  //Write params is SRAM
WriteIO(SRAM_PARAMS_BASE + 0x04, Params2); //Write params is SRAM

// Put row data into SRAM buffer
for (i = 0; i < ProtectionPerMacro; i += 4)
{
   Params1 = (Data[i] << 0) + (Data[i + 1] << 8) +
             (Data[i + 2] << 16) + (Data[i + 3] << 24);
   WriteIO(SRAM_PARAMS_BASE + 0x08 + i, Params1);//Write params is SRAM
}

//  Call "Load Latch" SROM API
WriteIO(CPUSS_SYSARG, SRAM_PARAMS_BASE); //Set location of parameters
WriteIO(CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_LOAD_LATCH);//Request SROM operation
Status = PollSromStatus();
if (!Status) return FAIL;

//4. Program protection setting of current macro into supervisory row.
Params = (SROM_KEY1 << 0) +//KEY1
         ((SROM_KEY2 + SROM_CMD_WRITE_PROTECTION) << 8) +//KEY2
         (ChipLevelProtection << 16)+    //Applicable only for Macro
         (Macro0 << 24);//Flash Macro

WriteIO(CPUSS_SYSARG, Params);
WriteIO(CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_WRITE_PROTECTION);

//Read status of the operation
Status = PollSromStatus();
if (!Status)) return FAIL;

return PASS;
//------------------------------------------------------------------------------
```

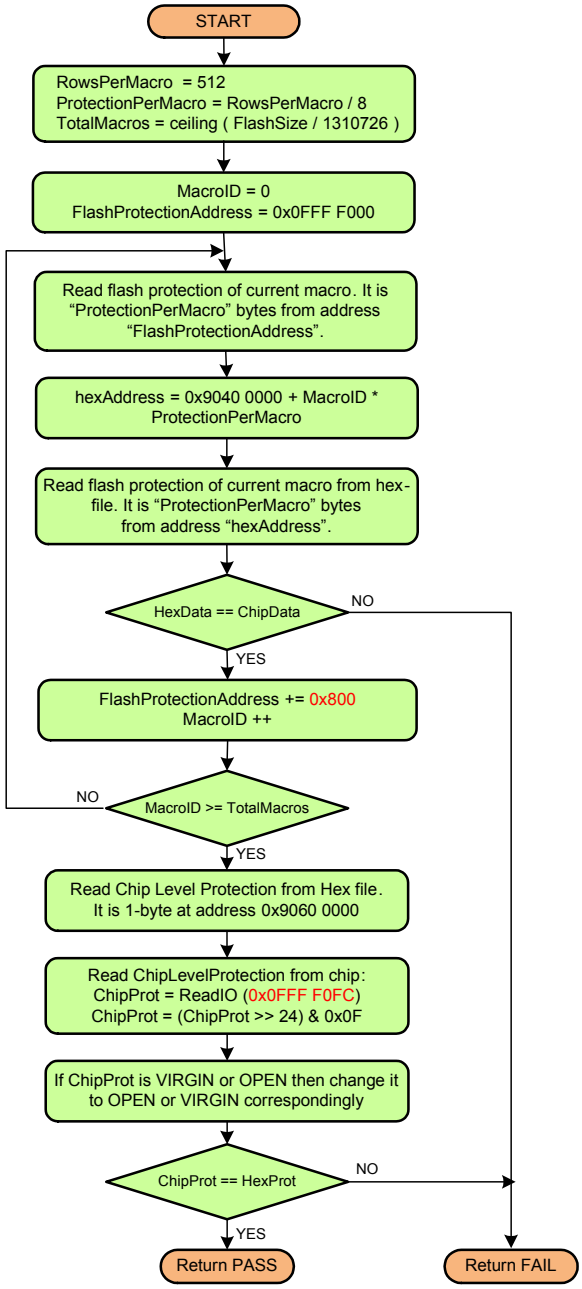
4.10 Step 8 – Verify Protection Settings

This step verifies the data that was written in Step 7. The point is to read back the details of flash protection and chip-level protection from the silicon and compare this data with the corresponding data from the hex file. Although this step is optional, Cypress recommends that you implement it in the programmer.

Reading of the protection setting is carried out by direct access to the memory space of the CPU (via AHB). See Figure 2-1 on page 7 to find the address range of protection data. The programmer reads out the data in 4-byte words.

Note that when a chip-level protection byte is read from the silicon, it must be reviewed. This is because of the inverted values of OPEN and VIRGIN modes written in the supervisory rows (see Appendix A:Chip-Level Protection on page 36). If VIRGIN mode is read from flash, then it must be converted to OPEN mode; if OPEN mode is read from flash, it must be considered as VIRGIN mode. For KILL and PROTECTED modes, no translation is necessary.

Figure 4-8. Flow Chart of the “Verify Protection Settings” Step



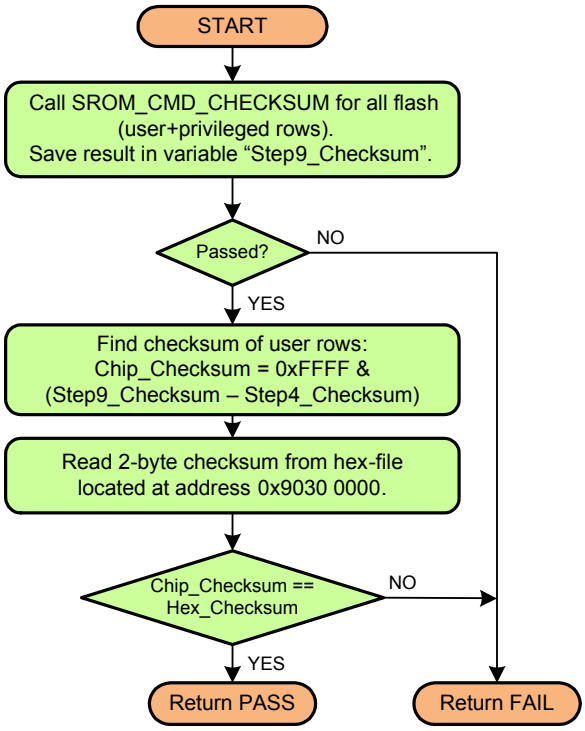
Pseudocode – Step 8. Verify Protection Settings

```
//-------------------------------------------------------------------------------
// Flash Size must be provided.
RowsPerMacro = 512;
ProtectionPerMacro = RowsPerMacro / 8;

TotalMacros = ceiling ( FlashSize /131072 ); // round to smallest following integer65536
FlashProtectionAddress = SFLASH_MACRO_0; //0x0FFF F000

for (MacroID = 0; MacroID < TotalMacros; MacroID++; FlashProtectionAddress += 0x800)
{
//1. Read Protection settings of current macro from hex-file.
//It is located at address 0x9040 0000.
//HEX_ReadRowProtection() must be implemented.
HexAddr = ProtectionPerMacro * MacroID;
hexProt = HEX_ReadRowProtection(HexAddr, ProtectionPerMacro);

//2. Read Protection of current macro from silicon
for (i = 0; i < ProtectionPerMacro; i += 4)
{
    ReadIO(FlashProtectionAddress + i, out data32);
    flashProt[i + 0] = (data32 >> 0) & 0xFF;
    flashProt[i + 1] = (data32 >> 8) & 0xFF;
    flashProt[i + 2] = (data32 >> 16) & 0xFF;
    flashProt[i + 3] = (data32 >> 24) & 0xFF;
}

//3. Compare hex and silicon's data
for (i = 0; i < ProtectionPerMacro; i++ )
{
    If (hexProt[i] != flashProt[i]) return FAIL;
}

//4. Read Chip Level Protection from hex-file. It is 1 byte at address 0x90600000.
//HEX_ReadChipLevelProtection() must be implemented.
Hex_ChipLevelProtection = HEX_ReadChipLevelProtection();

//5. Read Chip Level Protection from the silicon
ReadIO(SFLASH_CPUSS_PROTECTION, out Chip_ChipLevelProtection);
Chip_ChipLevelProtection = (Chip_ChipLevelProtection >> 24) & 0x0F;

if (Chip_ChipLevelProtection == CHIP_PROT_VIRGIN) Chip_ChipLevelProtection = CHIP_PROT_OPEN;
else
if (Chip_ChipLevelProtection == CHIP_PROT_OPEN) Chip_ChipLevelProtection = CHIP_PROT_VIRGIN;

//6. Compare hex's and silicon's data
if (Chip_ChipLevelProtection != Hex_ChipLevelProtection) return FAIL;

return PASS;
//-------------------------------------------------------------------------------
```

4.11 Step 9 – Verify Checksum

This step validates the result of the flash programming process. It calculates the checksum of the user rows written in Step 5 and compares this value with the 2-byte checksum from the hex file. The Checksum SROM API computes the checksum of the user and privileged rows. To find the checksum of only the user rows, it is necessary to subtract the checksum of the privileged rows found in Step 4. Figure 4-9 shows the final checksum algorithm. This is a mandatory step in the programming flow, although the checksum operation cannot completely guarantee that the data is written correctly. For this reason, the Verify Flash step is also recommended.

Figure 4-9. Flow Chart of the “Verify Checksum” Step



Pseudocode – Step 9. Verify Checksum

```
//------------------------------------------------------------------------------
// Checksum of Privileged rows must be taken from Step 4.
// SROM call here is identical to one Step 4, so it can be refactored into one subroutine.
// 1. SROM call - Checksum All
Params = (SROM_KEY1 << 0) +//KEY1
        ((SROM_KEY2+SROM_CMD_CHECKSUM) << 8)+//KEY2
        ((0x0000 & 0x00FF) << 16) +//Row ID[7:0]
        ((0x8000 & 0xFF00) << 16);//Row ID[15:8] - Checksum All(0x8000)

WriteIO( CPUSS_SYSARG, Params); //Write params in CPUSS_SYSARG
WriteIO( CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_CHECKSUM); //Request SROM call

status = PollSromStatus();
if (!status) return FAIL;

//Read Checksum from CPUSS_SYSARG register
ReadIO(CPUSS_SYSARG, out Checksum_all);

Checksum_All = (Checksum_All & 0x0FFFFFFF); //28-bit checksum

//2. Find 2-byte checksum of user rows, "Checksum_Privileged" is calculated in Step 4.
Chip_Checksum = (Checksum_All - Checksum_Privileged) & 0xFFFF;

//3. Read 2-byte checksum of user code from hex-file
//   HEX_ReadChecksum() must be implemented by Programmer.
Hex_Checksum = HEX_ReadChecksum();

//4. Compare silicon's vs hex's checksum
if (Chip_Checksum != Hex_Checksum) return FAIL;

return PASS;
//------------------------------------------------------------------------------
```

4.12 Step 10 - Program User SFlash (optional)

The Supervisory Flash of the Target device provides four rows in Macro 1 for application specific use. Every such row consists of 256 bytes - same as other flash rows. Application can store here any information, but typically it is the Unique Bluetooth Address of the device. Length and format of the address is also application specific. During mass production vendor should define the rule which guarantees that every programmed part is assigned with the unique address.

This step is considered as optional - every application should determine if it needs this flash region and for which purpose. Also, User SFlash rows are not stored in the hex file, since this information is unique for each part (for most of applications).

Programming of User SFlash via SWD port is only available in silicon's OPEN mode. So, we have to execute this step sometime after Erase All step, which guarantees that part is in OPEN mode. But, the recommendation is to execute it as the last step in the programming flow, since it's the non-hex flash region (optional). Alternatively, User Application can update this SFLash region whenever needed (CPU access via SROM APIs) - e.g. to store calibration data, non-volatile parameters, etc.

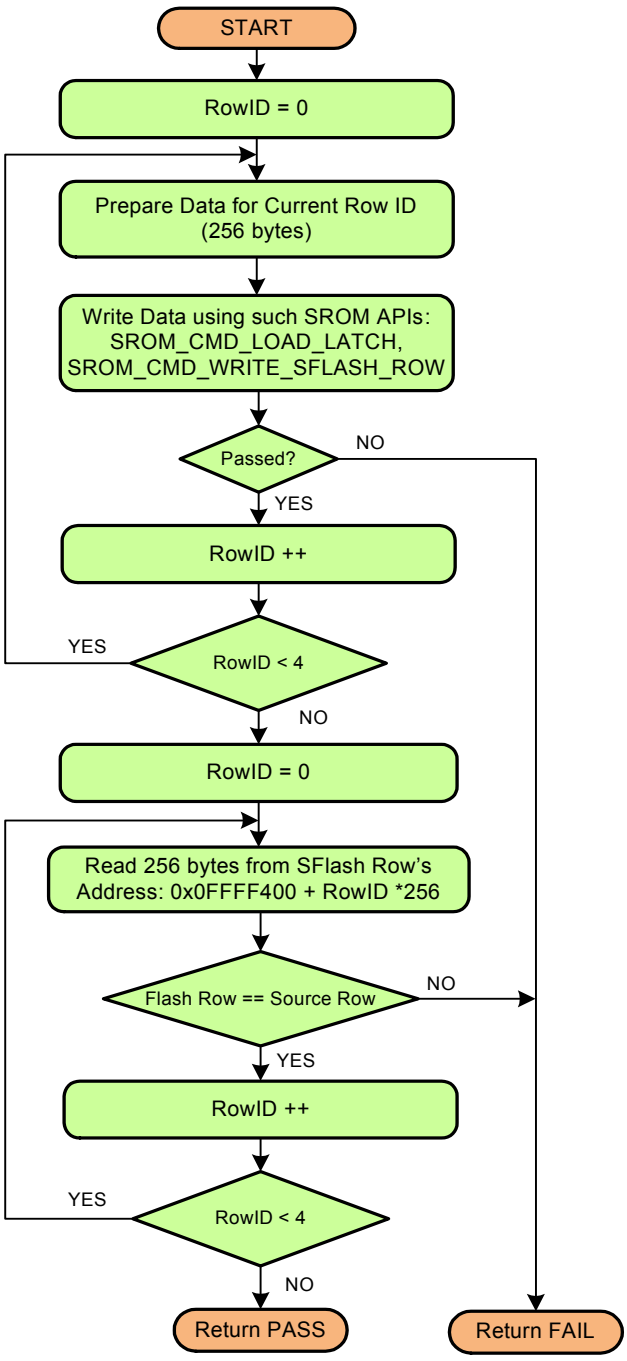
The User SFlash Rows are mapped onto CPUs address space in the range 0x0FFF F400 - 0x0FFF F7FF. So, user application can read these rows directly from these addresses.

Following SROM APIs are used in this step:

- SROM_CMD_LOAD_LATCH - Loads the flash row into the silicon's volatile buffer;
- SROM_CMD_WRITE_SFLASH_ROW - Program rows from volatile latch into User's Flash.

Figure 4-10 illustrates User SFlash programming algorithm.

Figure 4-10. Flow Chart of "Program User SFlash" Step



Pseudocode. Step 10 - Program User SFlash

```
//-------------------------------------------------------------------------------
// Flash Row Size, Number of Rows and Data to Program must be provided.
RowSize = 256;
TotalRows = 4;

//Program all User SFlash rows
for (int RowID = 0; RowID < TotalRows; RowID++)
{
    //1. Prepare data for current row (256-byte)
    //Read it in the "Data" array .
    //SFlash_ReadSource() must return data for current SFlash row.
Data = SFlash_ReadSource( RowID, RowSize );

    //2. Load Row to volatile buffer (latch)
    MacroID = 0x00; //User SFLash rows are located only in Macro 0

    Params1 = (SROM_KEY1 << 0) +//KEY1
          ((SROM_KEY2 + SROM_CMD_LOAD_LATCH) << 8) +//KEY2
          (0x00 << 16)+//Byte number in latch from what to write
          (MacroID << 24);       //Flash Macro ID (0 or 1)


    Params2 = (RowSize - 1); //Number of Bytes to load minus 1

    WriteIO(SRAM_PARAMS_BASE + 0x00, Params1);  //Write params in SRAM
    WriteIO(SRAM_PARAMS_BASE + 0x04, Params2); //Write params in SRAM

    //  Put row data into SRAM buffer
    for (i = 0; i < RowSize; i += 4)
    {
       Params1 = (Data[i] << 0) + (Data[i + 1 ] << 8) +
                 Data[i + 2] << 16) + (Data[i + 3] << 24);
       WriteIO(SRAM_PARAMS_BASE + 0x08 + i, Params1);//Write params is SRAM
    }

    //  Call "Load Latch" SROM API
    WriteIO(CPUSS_SYSARG, SRAM_PARAMS_BASE); //Set location of parameters
    WriteIO(CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_LOAD_LATCH);//Request SROM operation

    Status = PollSromStatus();
    if (!Status) return FAIL;

    //3. Program User SFlash Row - call SROM API
       Params1 =(SROM_KEY1 << 0) +  //KEY1
               ((SROM_KEY2+SROM_CMD_WRITE_SFLASH_ROW) << 8)+ //KEY2

    Params2 = RowID  //Row ID of User SFlash

    WriteIO(SRAM_PARAMS_BASE+0x00, Params1); //Write params is SRAM
    WriteIO(SRAM_PARAMS_BASE+0x04, Params2); //Write params is SRAM

    WriteIO(CPUSS_SYSARG, SRAM_PARAMS_BASE);    //Set location of parameters
    WriteIO(CPUSS_SYSREQ, SROM_SYSREQ_BIT | SROM_CMD_WRITE_SFLASH_ROW); //Request SROM oper-
ation

    Status = PollSromStatus();
```

```
    if (!Status) return FAIL;
}

//Verify all User SFlash rows
for (int RowID = 0; RowID < TotalRows; RowID++)
{
    //1. Prepare Source data for current row (256-byte)
    sourceData = SFlash_ReadSource( RowID, RowSize );

    //2. Read row from chip
    RowAddress = 0x0FFFF200 + RowID * RowSize;

    for (i = 0; i < RowSize; i += 4)
    {
        //Read flash via AHB-interface
        ReadIO( RowAddress + i, out data32);
        chipData[i + 0] = (data32 >> 0) & 0xFF;
        chipData[i + 1] = (data32 >> 8) & 0xFF;
        chipData[i + 2] = (data32 >> 16) & 0xFF;
        chipData[i + 3] = (data32 >> 24) & 0xFF;
    }

    //3. Compare them
    for (i = 0; i < RowSize; i++)
       {
       if (chipData[i] != sourceData[i]) return FAIL;
       }
}

return PASS;
```

Appendix A. Chip-Level Protection

The difference between chip-level protection and row-level protection may not be obvious at first glance. Chip-level protection restricts access to the silicon's resources by way of the SWD bus by the external programmer. However, it does not restrict anything for the firmware. If any resource is not accessible, the SWD transaction is NACKed. Row-level protection restricts the firmware and the external programmer from writing to the protected flash rows.

There are four states of chip-level protection into which the silicon can be moved: VIRGIN, OPEN, PROTECTED, KILL.

Table A-1. States of Chip-Level Protection

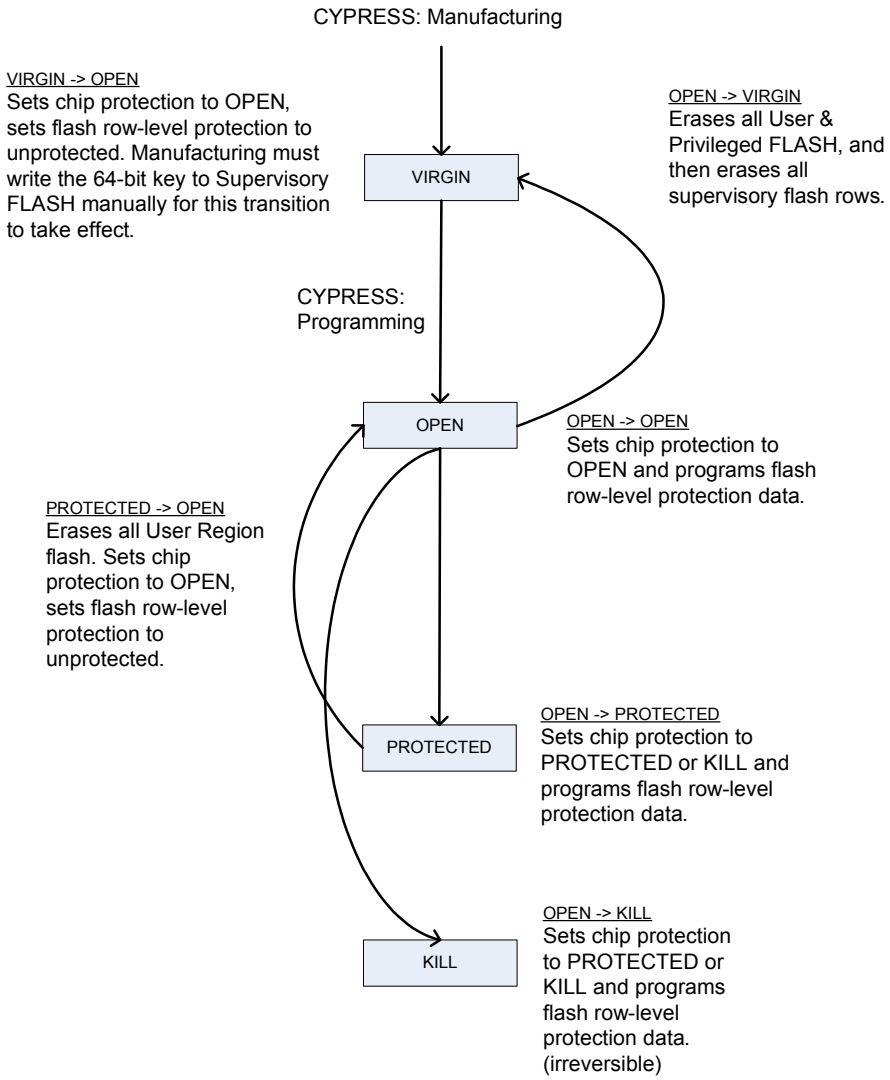
Protection State	Value in Hex and CPUSS_PROTECTION	Value in Written Supervisory Row	Restrictions
VIRGIN	0x00	0x01	In this mode, the silicon is in post-fab (untrimmed state). After trimming, the silicon is moved into OPEN mode for customer. This mode is not for custom use. Customers are not physically prohibited from bringing parts back to the VIRGIN state, but they will be left with parts missing critical trim, wounding, and other settings from Cypress. This essentially makes the part unusable for the customer.
OPEN	0x01	0x00	In this mode, the silicon is shipped to customers. Most applications use this state in which an external debugger can access all the needed resources for full-functional debugging of the application. Flash, SRAM, supervisory flash, and registers are available via the DAP.
PROTECTED	0x02	0x02	In this mode, the silicon allows limited access via the DAP; it is enough to read the silicon ID and move the chip back to OPEN mode. Access to Flash, SRAM, and most of the registers is disabled, so SWD transactions are NACKed for master. This is true for read and write requests on the SWD bus.
KILL	0x04	0x04	KILL mode completely locks the SWD-pins from an external programmer. The firmware must be 100 percent operable without bugs because it can no longer be updated. If this mode is needed, then it is recommended that you enable it only for production programming of end-application.

The chip-level protection byte is located in the supervisory row of the macro at offset 0x7F. It can be programmed only when row-level protection is updated for the macro. The actual value of the OPEN state that is written into flash is 0x00 and not 0x01, which is the real value in the hex file. For the VIRGIN and OPEN modes, the value saved in the supervisory row is inverted. This is done for one reason—to prevent accidental resets to the VIRGIN state during programming.

The EraseAll() operation clears a whole row, resetting every byte to 0. After the EraseAll() operation, which is the first operation targeting the flash during programming, the chip is left in the VIRGIN mode, which is not correct. It must be in OPEN mode even after the chip is reset. During startup, the boot code reads 0x00 from the supervisory row and translates it to 0x01 before writing to the CPUSS_PROTECTION register, which defines the current mode for the CPU. The corresponding value of 0x01 from the supervisory row is translated to 0x00 (VIRGIN) for CPUSS_PROTECTION. PROTECTED and KILL modes are not changed by the boot code and are copied directly to the CPUSS_PROTECTION register. Specifically, the OPEN-VIRGIN modes swapped in flash must be considered during the verification operation, when the protection byte is read from the supervisory row and compared with the corresponding value from hex.

The chip has a special policy of changing the state of chip-level protection; this means that the possible new state is dependent on the current protection state. See Figure A-1 on page 37 for possible transition paths.

Figure A-1. Chip-Level Protection State Diagram



The customer receives the device in the OPEN mode and can move it to OPEN, PROTECTED, or KILL. Moving to VIRGIN mode is discouraged because the part will be untrimmed and therefore not operable. From PROTECTED mote, the customer can move the part back to OPEN. There is no way to leave the KILL mode.

Appendix B. Intel Hex File Format

Intel hex file records are a text representation of hexadecimal-coded binary data. Only ASCII characters are used, so the format is portable across most computer platforms. Each line (record) of Intel hex file consists of six parts, as shown in Figure B-1.

Figure B-1. Hex File Record Structure

Start code, one character - an ASCII colon ':'

- **Byte count**, two hex digits (1 byte) - specifies the number of bytes in the data field.
- **Address**, four hex digits (2 bytes) - a 16-bit address of the beginning of the memory position for the data.
- **Record type**, two hex digits (00 to 05) - defines the type of the data field. The record types used in the hex file generated by Cypress are as follows.
  - 00 - Data record, which contains data and 16-bit address.
  - 01 - End of file record, which is a file termination record and has no data. This must be the last line of the file; only one is allowed for every file.
  - 04 - Extended linear address record, which allows full 32-bit addressing. The address field is 0000, the byte count is 02. The two data bytes represent the upper 16 bits of the 32-bit address, when combined with the lower 16-bit address of the 00 type record.
- **Data**, a sequence of 'n' bytes of the data, represented by 2n hex digits.
- **Checksum**, two hex digits (1 byte), which is the least significant byte of the two's complement of the sum of the values of all fields except fields 1 and 6 (start code ':' byte and two hex digits of the checksum).

Examples for the different record types used in the hex file generated for CYBL10x7x device are as follows.

Consider that these three records are placed in consecutive lines of the hex file (chip-level protection and end of hex file).

*:020000**04**9060**0**A*

*:01**0000**00**02**FD*

*:00000**01**ff*

For the sake of readability, "record type" is highlighted in red and the 32-bit address of the chip-level protection is in blue.

The first record (*:0200000490600A*) is an extended linear address record as indicated by the value in the Record Type field (04). The address field is 0000, the byte count is 02. This means that there are two data bytes in this record. These data bytes (0x9060) specify the upper 16 bits of the 32-bit address of data bytes. In this case, all the data records that follow this record are assumed to have their upper 16-bit address as 0x9060 (in other words, the base address is 0x90600000). 0A is the checksum byte for this record:

0x0A = 0x100 – (0x02+0x00+0x00+0x04+0x90+0x60).

The next record (:0100000002FD) is a data record, as indicated by the value in the Record Type field (00). The byte count is 01, meaning there is only one data byte in this record (02). The 32-bit starting address for these data bytes is at address 0x90600000. The upper 16-bit address (0x9060) is derived from the extended linear address record in the first line; the lower 16-bit address is specified in the address field of this record as 0000. FD is the checksum byte for this record.

The last record (:00000001FF) is the end-of-file record, as indicated by the value in the Record Type field (01). This is the last record of the hex file.

Note The data records of the following multibyte region in the hex file are in big-endian format (MSB in lower address): checksum data at address 0x9030 0000, metadata at address 0x9050 0000. The data records of the rest of the multibyte regions in the hex file are all in little-endian format (LSB in lower address).

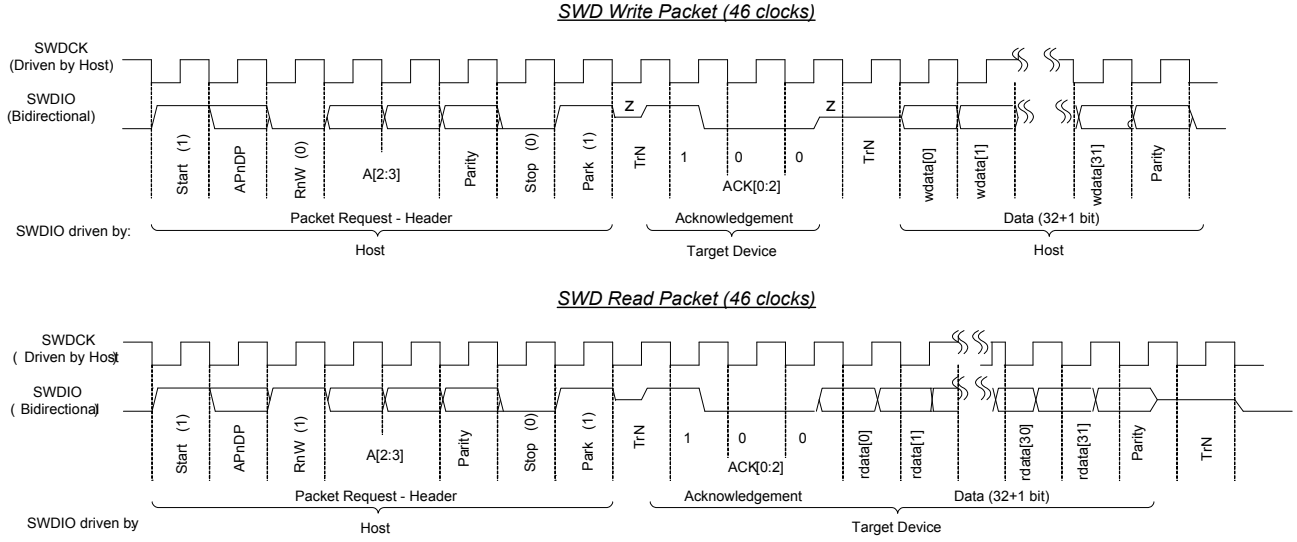
Appendix C. Serial Wire Debug (SWD) Protocol

The SWD protocol is a packet-based serial transaction protocol. At the pin level uses a single bidirectional data connection (SWDIO) and a clock connection (SWDCK). The host programmer always drives the clock line, while either the programmer or the target device drives the data line. A complete data transfer (one SWD packet) requires 46 clocks and consists of three phases:

- **Packet Request** – The host programmer issues a request to the target device (silicon).
- **Acknowledge Response** – The target device (silicon) sends an acknowledgement to the host.
- **Data Transfer Phase** – The data transfer is either from the target to the host, following a read request (RDATA), or from the host to the target, following a write request (WDATA). This phase is only present when a packet request phase is followed by a valid (OK) acknowledge response.

Figure C-1 shows the timing diagrams of the read and write SWD packets.

Figure C-1. Write and Read SWD Packet Timing Diagrams



a) Host Write Cycle – host sends data on the SWDIO line on falling edge of SWDCK and target will read that data on next SWDCK rising edge (for example, 8-bit header data).

b) Host Read Cycle – target sends data on SWDIO line on rising edge of SWDCK and the Host should read that data on next SWDCK falling edge (for example, ACK phase (ACK[2:0]), Read Data (rdata[31:0])).

c) The Host should not driver the SWDIO line during TrN phase. During first TrN phase (½ cycle duration) of SWD packet, target starts driving the ACK data on the SWDIO line on the rising edge of SWDCK. The host should read the data on the subsequent falling edge of SWDCK. The second TrN phase is 1.5 clock cycles as shown in figure above. Both target and host will not drive the line during the entire second TrN phase (indicated as 'z'). Host should start sending the Write data (wdata) on next falling edge of SWDCK after second TrN phase.

The SWD packet is transmitted in this sequence:

1. The start bit initiates a transfer; it is always logical ‘1’.
2. The APnDP bit determines whether the transfer is an AP access (indicated by ‘1’), or a DP access (indicated by ‘0’).
3. The next bit is RnW, which is ‘1’ for read from the device or ‘0’ for a write to the device.
4. The ADDR bits (A[3:2]) are register select bits for the access port or debug port. See Table 3-2 on page 11 for register definition.
5. The parity bit contains the parity of APnDP, RnW, and ADDR bits. This is an even parity bit. If the number of logical 1s in this bits is odd, then the parity must be ‘1’, otherwise it is ‘0’.

   If the parity bit is not correct, the target device ignores the header, and there is no ACK response. From the host standpoint, the programming operation should be aborted and retried by doing a device reset.
6. The stop bit is always logic ‘0’.
7. The park bit is always logic ‘1’ and should be driven high by the host.
8. The ACK bits are device-to-host response. Possible values are shown in Table C-1. Note that ACK in the current SWD transfer reflects the status of the previous transfer. OK ACK means that the previous packet was successful. WAIT response requires a data phase, as explained in the following list. For a FAULT status, the programming operation should be aborted immediately.
   a. For a WAIT response, if the transaction is a read, the host should ignore the data read in the data phase. The target does not drive the line and the host must not check the parity bit as well.
   b. For a WAIT response, if the transaction is a write, the data phase is ignored by the target device. However, the host must still send the data to be written from the standpoint of implementation. The parity data parity bit corresponding to the data should also be sent by the host.
   c. For a WAIT response, it means that the target device is processing the previous transaction. The host can try for a maximum four continuous WAIT responses to see if an OK response is received. If it fails, then the programming operation should be aborted and retried.
   d. For a FAULT response, the programming operation should be aborted and retried by doing a device reset.

Table C-1. ACK Response for SWD Transfers

ACK[2:0]	SWD
OK	001
WAIT	010
FAULT	100
NACK	111

9. The data phase includes a parity bit (even parity)
   a. For a read packet, if the host detects a parity error, then it must abort the programming operation and try again.
   b. For a write packet, if the target device detects a parity error in the data sent by the host, it generates a FAULT ACK response in the next packet.
10. Turnaround (TrN) phase: There is a single-cycle turnaround phase between the packet request and the ACK phases, as well as between the ACK and data phases for write transfers as shown in Figure C-1. According to the SWD protocol, both the host and the target use the TrN phase to change the drive modes on their respective SWDIO lines. During the first TrN phase after packet request, the target starts driving the ACK data on the SWDIO line on the rising edge of SWDCK in the TrN phase. This ensures that the host can read the ACK data on the next falling edge. Thus, the first TrN cycle lasts for only a half-cycle duration. The second TrN cycle of the SWD packet is one and one-half cycle long. Neither the host nor the target device should drive the SWDIO line during the TrN phase, as indicated by ‘z’ in Figure C-1.
11. The address, ACK, and read and write data are always transmitted LSB first.
12. According to the SWD protocol, the host can generate any number of SWD clock cycles between two packets with the SWDIO low. It is recommended that you generate several dummy clock cycles (three) between two packets or make clock free running in IDLE mode.

Note The SWD interface can be reset by clocking 50 or more cycles with the SWDIO kept high. To return to the idle state, SWDIO must be clocked low once.

Appendix D. Timing Specifications of the SWD Interface

The external host should perform all read or write operations on the SWDIO line on the falling edge of SWDCK. The target device performs read or write operations on SWDIO on the rising edge of SWDCK.

Figure D-1. SWD Interface Timing Diagram

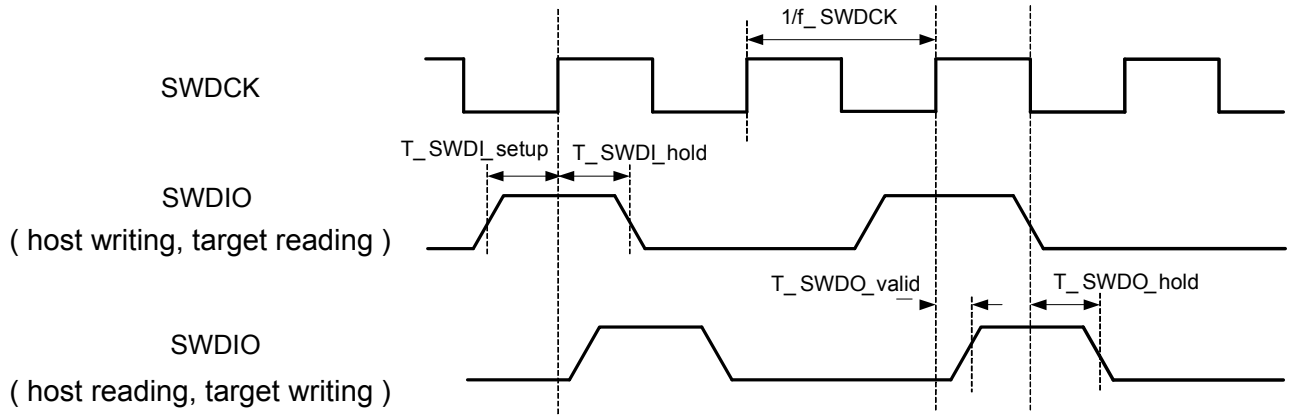


Table D-1. SWD Interface AC Specifications

Symbol	Description	Conditions	Min	Typ	Max	Units
f_SWDCK	SWDCLK frequency	3.3 V ≤ VDDD ≤ 5.0 V	–	–	14	MHz
		1.71 V ≤ VDDD ≤ 3.3 V	–	–	7	MHz
T_SWDI_setup	SWDIO input setup before SWDCK high	T = 1 / f_SWDCK	T/4	–	–	ns
T_SWDI_hold	SWDIO input hold after SWDCK high	T = 1 / f_SWDCK	T/4	–	–	ns
T_SWDO_valid	SWDCK high to SWDIO output valid	T = 1 / f_SWDCK	–	–	T/2	ns
T_SWDO_hold	SWDIO output hold after SWDCK high	T = 1 / f_SWDCK	1	–	–	ns

Although the ARM specification does not define the minimum frequency of the SWD bus, the minimum for the Target family is 1.5 MHz. It is only needed on the first step to acquire the silicon during the boot window. After that, the programming frequency can be as low as needed.